ME 120: `while` Loops in Arduino

July 9, 2019

Gerald Recktenwald

gerry@pdx.edu

1 Overview

A `while` loop structure is an alternative to a `for` loop for computations that need to be repeated. In this document, the syntax of the `while` loop is described and a series of simple example programs demonstrate how `while` loops work. The article ends with a practical example of using a `while` loop to wait for an external events such as button input or an analog input reaching a threshold.

A `while` loop is an iteration structure that is most often used when there is a unknown number of repetitions to be performed. A `while` loop may have, but does not need, a loop counter. A `while` loop is a natural way to repeat an operation until a condition is met, especially when that condition is not directly or obviously dependent on the number of repetitions of the loop. The generic structure of a while loop is

```
while ( Continuation condition  ) {
    Loop body
}
```

2 `while` Loop Syntax

In this section we describe the syntax of the the `while` loop structure. Your Arduino code can have as many (or as few) loops as you need. Remember that the `loop` function is itself executed repeatedly. When using a `while` loop structure you are introducing an additional form of repetition.

2.1 `while` loop

Figure 1 shows an annotated `while` loop that waits for a random number larger than 5 to be generated by the built-in `random` function. Note that this loop does not have a specified number of iterations to be executed. Because the sequence of numbers returned by a sequence of calls to `random` is unpredictable, the number of repetitions of the loop will vary as the code is repeated.

The behavior of a `while` loop is controlled by the continuation condition. Typically the continuation condition is a simple logical expression. It would seem, therefore, that the `while` loop does its job with less need for code than a corresponding `for` loop. In fact, there is no real code savings in using a `while` loop because important statements that affect the continuation condition are not inside the parenthesis immediately following the `while` statement. The code in Figure 1 shows that the `while` loop has start and stop conditions analogous to the start/stop conditions of a `for` loop.

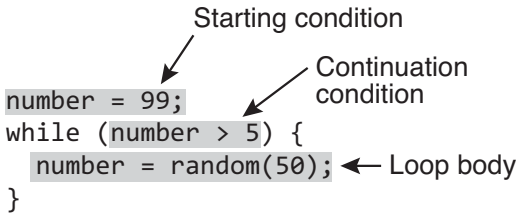

Figure 1: Components of the counter specification of a `while` loop.

Example: Compute $\sum_{i=1}^{10} i$

`while` loops can be used instead of `for` loops with an appropriate change of the continuation criteria. For example, the following blocks show how to add the numbers from 1 to 10 with a `for` loop on the left, and a `while` loop on the right.

```
int i, sum;

sum = 0;
for ( i=0; i<=10; i++ ) {
  sum = sum + i;
}
```

```
int i, sum;

i = 0;
sum = 0;
while ( i<=10 ) {
  i = i + 1;
  sum = sum + i;
}
```

The preceding computation is more naturally expressed with a `for` loop, because we know how many iterations are needed prior to the start of the loop. Furthermore, the `while` structure appears more verbose due to the need to initialize and increment the counter variable, `i`, outside of the `while` statment itself. Although it is not wrong to use a `while` loop to sum the numbers, the `for` loop does the same operations more compactly. There are other applications where a `while` loop is more natural.

At a logical level, a `for` loop and a `while` loop are equivalent. At a practical level the choice of a `for` loop or `while` loop is best evaluated according to the programmers intention or reason for writing a loop. Use a `for` loop if you need to iterate a number of times that is known at the start of the loop. Use a `while` loop if you don't know how many iterations are necessary, and especially if the condition for stopping the iterations are determined by some external event, like the pressing of a button.

2.2 Run-time Errors in `while` loops

As with any programming feature, it is possible to make errors in logic that produce intended effects. For example, consider the three code snippets that show what happens when important statements are not included.

Infinite loop:

```
int i, sum;

sum = 0;
while ( i<=10 ) {
  sum = sum + i;
}
```

Ambiguous loop:

```
int i, sum;

sum = 0;
while ( i<=10 ) {
  i = i + 1;
  sum = sum + i;
}
```

Well-defined loop:

```
int i, sum;

i = 0;
sum = 0;
while ( i<=10 ) {
  i = i + 1;
  sum = sum + i;
}
```

The code on the left is an infinite loop – once started it never terminates because the value of `i` is never changed. The code in the middle is ambiguous because the initial value of `i` is not specified. Depending on the compiler and compiler settings used to convert the source code to an executable binary code, the value of `i` may be set to zero, or it may simple take on the value defined by the bit pattern in RAM when the code starts. In other words, the code in the middle is ambiguous and

unreliable because the same code may result in different outcomes depending on how that code is converted from human-readable form to machine-executable form. The code on the right is well-defined and should run the same regardless of the compiler settings or the hardware running the code.

The preceding examples show that proper use of a `while` loop requires the programmer to prepare for the first evaluation of the starting condition. The programmer must write the continuation condition to specify when the loop terminates. The programmer must also specify some mechanism for changing the outcome of the continuation condition.

3 Examples

In this section a series of very simple Arduino sketches is discussed. Read the sketch and predict its behavior *before* you read the explanation provided here.

This section provides a series of very simple Arduino sketches to demonstrate how `for` loops work. The codes do not perform useful functions. Refer to Section **??** for an scenario where a `for` loop performs a useful function.

> **Note to instructors and to students using these notes for self-study:**
> Explanations are given here to support instructors, or students doing self-study. It would be best for students to study these codes, and predict the outcome of running the code *before* consulting the notes in the right-hand column and *before* they use the Arduino to show how the code actually works.

Loop 1

```
void setup() {
  Serial.begin(9600);
}

void loop() {
  int i=0;
  while ( i<5 ) {
    Serial.println(i);
  }
  Serial.println("... end while\n");
}
```

What is the output of the code to the left?

Answer: This code compiles and runs. However, the code gets stuck in the `while` loop because the value of `i` never changes. This is an example of an *infinite* loop

Loop 2

```
void setup() {
  Serial.begin(9600);
}

void loop() {
  int i=0;
  while ( i<5 ) {
    Serial.println(i);
    i = i + 1;
  }
  Serial.println("... end while\n");
}
```

What is the output of the code to the left?

Answer: This code compiles and runs. Every time the `loop` function is executed, the values 0, 1, 2, 3, 4 are printed on separate lines, followed by the `... end while` message.

Loop 3

```
void setup() {
  Serial.begin(9600);
}

void loop() {
  int i=0;
  while ( i<5 ) {
    Serial.println(i);
    i = i + 2;
  }
  Serial.println("... end while\n");
}
```

What is the output of the code to the left?

Answer: This code compiles and runs. Every time the `loop` function is executed, the values 0, 2, 4 are printed on separate lines, followed by the `... end while` message.
Extra question: what happens if the test for continuing is changed to `while ( i<=5 )`? In other words, what happens when `i<5` is changed to `i<=5` in the conditional test of the loop?

Loop 4

```
void setup() {
  Serial.begin(9600);
}

void loop() {
  int i=0, n=7;
  while ( i<n ) {
    Serial.print(i);  Serial.print("  ");
    i = i + 1;
  }
  while ( i>=0 ) {
    Serial.print(i);  Serial.print("  ");
    i = i - 1;
  }

  Serial.println("");
}
```

What is the output of the code to the left?

Answer: This code compiles and runs. Every time the `loop` function is executed, the values 0, 1, 2, ..., 8, 7, ..., 2, 1, 0 are printed on separate lines.

```
void setup() {
  int button_pin = 9;           //  Digital input on pin 9
  int button_pressed = FALSE;   //  Status of the button, initially not pressed
  pinMode( button_pin, INPUT);  //  Configure the input pin

  // Repeat until the user presses a button connected to button_pin
  while ( !button_pressed ) {
     button_pressed = digitalRead( button_pin );
  }
}

void loop() {

   // -- Do useful stuff after button press in setup() allows execution to continue
}
```

Listing 1: Using a `while` loop to wait for a button press.

4 A Practical Examples

Suppose you want to suspend all program activity until a specific event occurs. Two practical examples are

- Wait for button press, or
- Wait for a system to warm up or a tank to fill up or drain

These two examples *block* execution until an event occurs. While waiting – and testing for the even to occur, no other program tasks are completed. You can rearrange code to avoid blocking, but we will start with the simple approach of blocking execution.

4.1 Waiting for a Button Press

The code in Listing 1 shows a more natural use for a `while` loop, namely, waiting in the `setup` function for a button to be pushed. That would be useful if you wanted to make your program wait for some condition to be manually established before the `loop` function is to begin. For example, suppose a user wants to run a test that requires a human to set up the physical hardware before the test starts. The Arduino program would be designed to wait for a button press before starting the test.

4.2 Waiting for Input Threshold

The code in Listing 2 is a variation on the wait-for-button press. Instead of waiting for user input via a button, this example uses a sensor readings to block execution. An example would be a system that needs to "warm up" in some way.

The wait-for-event logic could be applied anywhere in an Arduino program, not just at the start of the program, as in Section 4.1, or waiting for the system to warm up. For example, suppose a laboratory process required a tank to have a minimum level of liquid at all times. A sensor could be used to measure the liquid level. Whenever the tank level was below the required minimum, a wait-until-the-tank is full algorithm could open a valve and monitor the tank level.

```
void setup() {
  int sensor_pin = 1;            //  Analog input pin
  int reading, threshold = 500;  //  reading holds current sensor value,
                                 //  threshold is limit to wait for

  // -- Repeat until the sensor reading exceeds a threshold
  reading = 0;  //  initialize reading so that while condition is met on first try

  while ( reading<threshold ) {           // Continue until reading exceeds threshold
     reading = analogRead( sensor_pin );
  }
}

void loop() {

   // -- Do useful stuff after sensor exceeds threshold, allowing execution to continue
}
```

Listing 2: Using a `while` loop to wait for a change in sensor status. Other sensor threshold criteria could be used. Here we demonstrate a simple one.